HAL
open science

Hardware acceleration of sequential loops

Pierre Michaud

To cite this version:

Pierre Michaud. Hardware acceleration of sequential loops. [Research Report] RR-7802, INRIA. 2011. hal-00641350

HAL Id: hal-00641350

https://inria.hal.science/hal-00641350v1

Submitted on 15 Nov 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Hardware acceleration of sequential loops

Pierre Michaud

RESEARCH REPORT

N° 7802

November 2011

Project-Teams ALF

ISSN 0249-6399 ISRN INRIA/RR--7802--FR+ENG

Hardware acceleration of sequential loops

Pierre Michaud

Project-Teams ALF

Research Report n° 7802 — November 2011 — 19 pages

Abstract: The current trend in general-purpose microprocessors is to take advantage of Moore's law to increase the number of cores on the same chip. In a few technology generations, this will lead to chips with hundreds of superscalar cores. Obtaining high performance on these so-called many-cores will require to parallelize the applications. Nevertheless, it is unlikely that all the applications will take full advantage of the high number of cores. Hence it is important, along with increasing the number of cores, to increase sequential performance and dedicate a relatively large silicon area and power budget for that purpose. In this study, we consider the possibility to increase sequential performance with a loop accelerator. The loop accelerator sits beside a conventional superscalar core and is specialized for executing dynamic loops, i.e., periodic sequences of dynamic instructions. Loops are detected and accelerated automatically, without help from the programmer or the compiler. The execution is migrated from the superscalar core to the loop accelerator when a dynamic loop is detected, and back to the superscalar core when a loop exit condition is encountered. We describe the proposed loop accelerator and we study its performance on the SPEC CPU2006 applications. We show that significant performance gains may be achieved on some applications.

Key-words: Multi-core processor, loop accelerator, sequential performance

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Accélération matérielle de boucles séquentielles

Résumé : La tendance actuelle des microprocesseurs généralistes est d'exploiter la loi de Moore en augmentant le nombre de coeurs sur une même puce. Dans quelques générations technologiques, cette tendance produira des puces avec des centaines de coeurs superscalaires. Il sera nécessaire de paralléliser les applications afin d'obtenir de hautes performances sur ces futurs multi-coeurs. Cependant, il ne sera probablement pas possible pour toutes les applications d'exploiter tous les coeurs. Il est donc important de continuer d'augmenter la performance séquentielle en même temps que la performance parallèle, en réservant à l'accélération séquentielle une partie relativement importante de la surface de silicium et du budget en puissance électrique de la puce. Dans cette étude, nous considérons la possibilité d'augmenter la performance séquentielle grâce à un accélérateur de boucle. L'accélérateur de boucle est associé à un coeur superscalaire classique et est spécialisé pour l'exécution des boucles dynamiques, c'est-à-dire des séquences périodiques d'instructions dynamiques. Les boucles sont détectées et accélérées automatiquement, sans aide du programmeur ou du compilateur. L'exécution migre du coeur superscalaire vers l'accélérateur de boucle quand une boucle dynamique est détectée, et vice versa lorsqu'on rencontre une condition de sortie de boucle. Nous décrivons l'accélérateur de boucle proposé et nous étudions sa performance sur les applications SPEC CPU2006. Nous montrons que des gains de performance relativement importants peuvent être obtenus pour certaines applications.

Mots-clés : Processeur multi-coeur, accélérateur de boucle, performance séquentielle

1 Introduction

During the last decade, single-thread performance has increased at a slower pace than during previous decades, despite transistor miniaturization continuing as dictated by Moore's law. For several reasons, processor makers have preferred to use the silicon area offered by miniaturization for implementing multi-cores. General-purpose multi-cores have actually benefited to the Internet by increasing the throughput of server farms. However, the client side of the internet as not benefited as much from multi-cores as the server side. While multi-cores have been definitely useful for increasing throughput, their potential for decreasing latency is still largely underexploited. Most existing code is sequential, and this is likely to remain so for some time. Writing parallel applications with *portable* performance speedups is very difficult, even for the elite of programmers who understand performance and know parallel programming. The gap between potential and actual performance will get larger as the number of on-chip cores increase, because of limited off-chip memory bandwidth and because of Amdahl's law. As the number of on-chip cores keeps increasing with years, we will progressively enter the *many-core* era and eventually reach a point where implementing a heterogeneous many-core with one big and fast core and several smaller cores will provide a significant performance advantage. For instance, let us consider a homogeneous many-core with 200 identical *normal* cores on one hand, and on the other hand a heterogeneous many-core with only 100 normal cores and one *monster* core taking the silicon area and consuming the power equivalent to 100 normal cores. Even if the monster core is only twice faster than a normal core despite being 100 times bigger, a simple application of Amdahl's law show that the heterogeneous many-core will outperform the homogeneous core on programs whose sequential fraction exceeds 1% of all the instructions executed [1]. This extreme example illustrates a situation that has been analyzed by Hill and Marty [6], one of their conclusions being that *"researchers should investigate methods of speeding sequential performance even if they appear locally inefficient"*.

What this means is that researchers must find solutions for accelerating sequential execution for future many-cores even if these solutions look absurd for today's multi-cores. However, increasing sequential performance is not obvious, even with the silicon area and power budget of a monster core. Increasing the IPC (number of instuctions executed per cycle) without impacting the clock cycle is not straightforward. Perhaps it will be possible to implement wider superscalar cores, e.g., 8-wide cores like the canceled Alpha EV8 processor. But to the best of our knowledge it has not been proved that the superscalar width could be increased beyond 8 without actually losing some performance. The problem comes from structures that do not scale well with the issue width and the instruction window size, like register renaming, dynamic instruction scheduling, register ports, operand bypass network, and memory disambiguation mechanisms. Many propositions for solving these problems have been published in the last 20 years, and some of them are probably worth revisiting in the context of many-cores. Nevertheless, it is also important to explore new approaches.

We propose in this study to explore a new approach to sequential performance : hardware acceleration of loops. Many programs spend a significant part of the execution in *dynamic* loops, i.e., periodic sequences of dynamic instructions. We explore in this study the possibility to implement an accelerator specialized for dynamic loops and that does not require any help from the programmer or the compiler.

Our goal is not to explore the design space of loop accelerators, which we believe is huge. Instead, we tried to imagine an accelerator microarchitecture exploiting loop properties and avoiding as much as possible the usual superscalar bottlenecks. We have simulated the proposed loop accelerator and we show that it can potentially deliver a high sustained IPC. Actually, we believe that the microarchitecture we propose is scalable and can be pipelined at a clock frequency at least as high as the superscalar core frequency.

This work makes the following contributions : **(1)** We propose a hardware mechanism for detecting dynamic loops ; **(2)** We provide a characterization of the dynamic loop behavior of SPEC CPU2006 benchmarks and we show that a significant fraction of the dynamic loops have a large loop body consisting of several tens of instructions ; **(3)** We propose a loop accelerator that can accelerate most dynamic loops with a small or large body and that can issue simultaneously up to 32 μops out of program order from a window of several thousands of μops, with a local hardware complexity not greater than that of a conventional superscalar core ; **(4)** We propose solutions for memory disambiguation in a window of several thousands of μops, exploiting dynamic loop properties ; **(5)** We show that a significant fraction of dynamic μops are

[1] This is assuming perfect parallelism. Limited off-chip memory banwidth, various overheads and imperfect load balancing will make the situation worse for the homogeneous many-core.

clock frequency : 3 GHz ; **decode/rename** : 1 inst (any) + 3 "simple" insts (1 or 2 μops) ; **reorder buffer** : 64-μops; **load queue :** 32 loads ; **store queue :** 16 stores ; **dispatch** : 6 μops, dependency-based steering ; **schedulers** : 4 8-μop int, 2 8-μop FP/SSE, 2 16-μop loads ; **execution** : 4 int (1 mul or 1 div), 2 FP/SSE (1 div), 2 loads/stores ; **retirement** : 6 μops; **post-retirement** : 16-store post-retirement queue, 2 stores/cycle ; **branch predictor** : 64-Kbit TAGE, 18-Kbit ITTAGE [12] ; **branch misprediction** : recovery at execution, 12 cycles minimum penalty ; **cache line** : 64 B ; **IL1 cache** : 32 KB, 8-way asso, 1 line/cycle bandwidth, up to 6 pipelined misses; **DL1 cache** : 32 KB, 8-way asso, write back, 2 cycles latency, 8 banks, 8-byte bank width, virtually indexed, PC-based stride prefetch ; **DL1 misses** : 16 MSHRs; **L2 cache** : 512 KB, 8-way asso, DIP policy [10], write back, 9 cycles latency, 1 line/cycle bandwidth, stream prefetch [16] ; **L3 cache** : 8 MB, 16-way asso, DIP policy [10], write back, 18 cycles latency, 1 line/cycle bandwidth, stream prefetch [16] ; **memory+bus** : 210 cycles latency, 16 bytes/cycle bandwidth ; **ITLB** : 64 entries, 4-way ; **DTLB** : 64 entries, 4-way ; **TLB2** : 512 entries, 4-way, 4 cycles latency ; **page size** : 4 MB ; **load/store dependencies** : store sets [2] + single-entry forwarding from store queue [8] ;

Table 1: Baseline superscalar core.

actually redundant and can be removed from the dynamic loops.

This paper is organized as follows. Section 2 discusses prior work. We describe our simulation set-up in Section 3. Section 4 describes our loop detector and provides statistics about dynamic loops in the SPEC CPU2006 applications. Section 5 gives a detailed description of the proposed LA and provides a performance evaluation. Finally, section 6 concludes this study and gives some directions for future work. This paper uses many definitions and acronyms, they are listed in Table 5 at the end of the paper.

2 Related work

Kobayashi found that many programs spend a significant fraction of the execution in dynamic loops, especially scientific programs [7]. Our definition of dynamic loops is close to Kobayashi's one. Tubella and González described a hardware mechanism for the automatic detection of dynamic loops [17]. Their definition of dynamic loops is different from ours and targeted toward speculative multithreading. Some recent processors have a loop buffer to decrease the energy consumption by avoiding re-fetching and re-decoding the same loop body again and again. For instance, the Intel Nehalem has a Loop Stream Detector that can hold up to 28 μops [4]. García et al. describe a loop buffer that implements a register renaming mechanism for loops, more efficient than conventional register renaming hardware [5]. Stitt et al. proposed a LA for a system-on-chip, implemented with some configurable logic, able to detect and accelerate loops transparently [14]. The time for analyzing the loop and configuring the LA appears to be very long (tens of millions of clock cycles for relatively simple loops [14]). It is not clear whether this approach could be used in general-purpose systems. Clark et al. described an approach where loops are identified and modulo-scheduled dynamically (hence preserving binary compatibility) onto a LA decoupling memory accesses from computations [3]. Their LA has a *configurable compute accelerator* which can execute up to 15 integer operations in only 2 clock cycles. Vajapeyam et al. proposed a *dynamic vectorization* (DV) scheme [18] based on *trace processors* [19, 11]. This DV scheme has a few similarities with our loop accelerator (e.g., use of FIFO queues), but is overall very different. The DV scheme assumes 64 processing elements (PEs), different instances of a loop iteration being processed by different PEs. Neither memory disambiguation nor problems concerning communication bandwith and latency between PEs are addressed in [18].

3 Simulation set-up

Our baseline configuration simulates a modern x86 superscalar core. Our simulator is trace driven (we do not simulate wrong path effects). We used Pin 2.8 [9] to generate a trace for each SPEC CPU2006 benchmark. We compiled benchmarks for x86-64 architecture with gcc 4.4.3 "-O2". Each trace consists of 40 samples of 50 millions consecutive instructions, for a total of 2 billions dynamic instructions (and the associated memory references). Samples are regularly spaced so as to be representative of the whole benchmark execution. Some

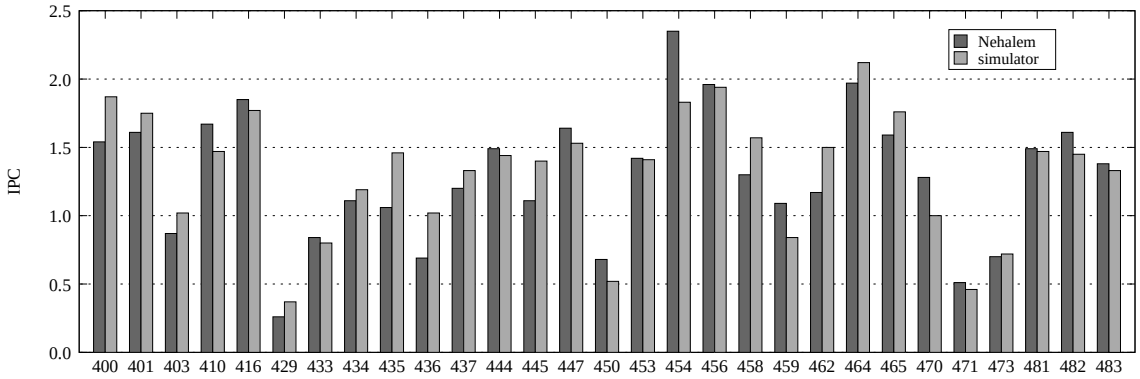

Figure 1: IPC of the simulated core vs. IPC measured on an Intel Nehalem for the SPEC CPU2006 benchmarks.

parameters of the baseline microarchitecture are listed in Table 1. Instructions are split into micro-ops (μops for short) at decode. We generate separate ADDR μops for memory accesses [8]. Load and store μops get the address from the associated ADDR μop, which is executed by any ALU. At most 8 μops are generated per instruction. A μop has at most 2 source registers and 1 destination register, not counting the flags. There are 4 integer schedulers, 2 floating-point schedulers and 2 load schedulers. Each scheduler can select one ready μop per cycle in its issue buffer. The μops is removed from the buffer when issued. We assume a rescheduling mechanism, but we did not simulate rescheduling penalties. We simulate hardware prefetchers for the DL1, L2 and L3 caches. The DL1 prefetcher is a PC-based stride prefetcher. The L2 and L3 prefetchers are stream prefetchers that issue prefetch requests based on the sequence of misses and hits on prefetched blocks [16]. The prefetch distance is adjusted dynamically with a feedback mechanism. We assume 4 MB memory pages. Using large pages decreases the number of TLB misses and makes prefetching in the L2 and L3 caches more effective, as stream prefetchers are limited by page boundaries [16]. TLB misses are hardware-managed, as in x86 processors [1].

The baseline **IPC** (instructions per cycle) numbers for the SPEC CPU2006 are reported in Figure 1. We also report the IPC measured on an Intel Nehalem processor when executing each benchmark to completion. The simulated microarchitecture is not identical to a Nehalem. For instance, we do not simulate μop fusion and macro-op fusion. Our goal was to simulate a realistic superscalar core, not to match the Nehalem perfectly.

4 The loop detector

In this study, the term *loop* is used in the sense of *dynamic loop*. A dynamic loop is a periodic sequence of dynamic instructions [7]. Instructions with different program counter addresses (**PC**, for short) are considered distinct, although they may perform the same action. The *loop length* is the number of dynamic instructions in the sequence. The *body size* B, in instructions, is the length of the smallest period. The *loop body* (or body, for short) consists of the first B instructions in the sequence, i.e., iteration number 0. The next B instructions belong to iteration number 1, and so on. Instructions in the body are not necessarily distinct. For example, the body itself may include a tiny loop unrolled, or a function executed multiple times. The loop detector is a hardware mechanism whose goal is to find loops. Because there will be a *transition penalty* for entering and leaving the loop acceleration mode, we try to detect loops that are long enough. The loop detector consists of two main parts : the Loop Monitor and the Loop Table.

The loop monitor. The Loop Monitor (**LM**) is the main mechanism for finding loops. It includes a tiny LM table (e.g., 4 entries). Each LM entry contains an end-of-loop PC (**EOL PC**) which is the search key, a valid bit, a body size, a body signature, a previous signature, an instruction count and a first-iteration bit. The LM table is fully-associative. For each instruction retiring from the reorder buffer, the body size of each valid LM entry is incremented and the corresponding body signature is updated. The signature is a hash of instructions PCs. It is intended to provide a unique identifier for each loop body encountered

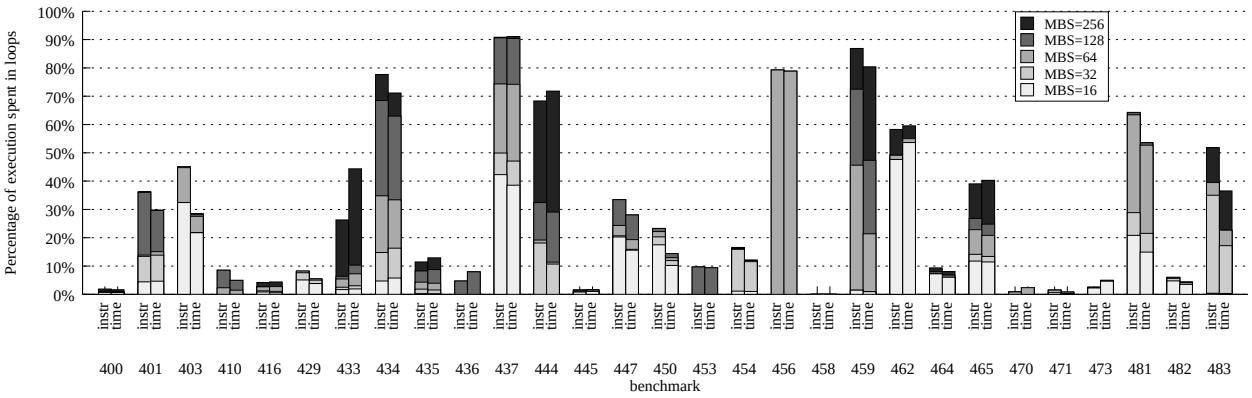

Figure 2: Percentage of execution (instructions and time) spent in loops (MinLM=900 instructions, LBI=5)

during the program execution. The longer the signature, the less likely that two distinct loop bodies have the same signature. In our simulations, we use 32-bit signatures, and we update the signature by rotating the signature 1 bit to the left and applying an XOR with the new instruction PC. When the body size of any valid LM entry exceeds a fixed *maximum body size* (**MBS**), we *close* that entry by resetting its valid bit. If the PC of a retiring instruction hits in the LM table, we look at the information stored in the matching entry. If the first-iteration bit is set or if the signature equals the previous signature, we add the body size to the instruction count, otherwise we reset the instruction count. When a branch instruction jumps backward, it is considered a potential end-of-loop. If no valid entry exists for that branch PC, we create a new entry (e.g., by reusing a closed entry) : the EOL PC is set to the branch PC, the valid bit and first-iteration bit are set, and all the other fields are reset. If an entry already exists for the backward jump, we copy the body signature into the previous signature and we reset the first-iteration bit, the body signature and the body size. When the instruction count in one of the LM entries exceeds a predefined threshold **MinLM**, we close all the other entries, and we enter the *loop building* state. The only valid LM entry remaining is called the active LM entry. Loop building corresponds to a fixed number of loop iterations during which we extract the information that will be needed to execute the loop on the LA. The number of iterations spent in the loop building state is denoted **LBI** (loop build iterations). When loop building is done, we migrate the execution to the LA. After the loop exit, we update the instruction count in the active LM entry and we close the entry.

The loop table. It takes MinLM dynamic instructions before an instruction sequence is considered a loop by the LM. On the one hand, if MinLM is fixed to a low value, short loops will be detected. But the execution time saved on the LA may not be worth the transition penalty. On the other hand, if MinLM is fixed to a high value, the MinLM instructions that could have been executed on the LA are instead executed by the superscalar core, which is a probable waste of performance. So MinLM is typically set to a medium value, e.g., 900 instructions. Yet, the next time we encounter this loop, we may assume that it will behave as the last time and enter the loop acceleration mode immediately. This is one of the functions of the *loop table* (**LT**). Each LT entry records some information about one loop, identified by its body signature. In particular, there is a *confidence* bit in each LT entry. When we close an LM entry, we set or reset the confidence bit depending on whether the instruction count is greater or less than MinLM. When the PC of a retired instruction matches one of the valid LM entries, we access the corresponding LT entry. If that LT entry exists and if the confidence bit is set, we close all the other LM entries and we enter loop building immediately.

4.1 Fraction of execution spent in loops

Figure 2 shows the fraction of execution spent in loops for different values of MBS. Two percentages are given for each benchmark, a percentage of executed instructions and a percentage of execution time. MinLM is set to 900 instructions and LBI=5. Several benchmarks spend a significant fraction of the execution in loops, but this is very dependent on MBS. If we execute all SPEC CPU2006 benchmarks to completion and

benchmark	MBS=128	MBS=256	benchmark	MBS=128	MBS=256
401.bzip2	5231	4838	450.soplex	3216	3261
403.gcc	8315	8312	456.hmmer	5446	5448
433.milc	2190425	6415123	459.GemsFDTD	4249	4953
434.zeusmp	4257	4359	462.libquantum	15362	18216
437.leslie3d	2218	2223	465.tonto	4161	4027
444.namd	3749	6320	481.wrf	2643	2645
447.dealII	2634	2627	483.xalancbmk	4239	5475

Table 2: Average loop length for loopy benchmarks (MinLM=900 instructions, LBI=5)

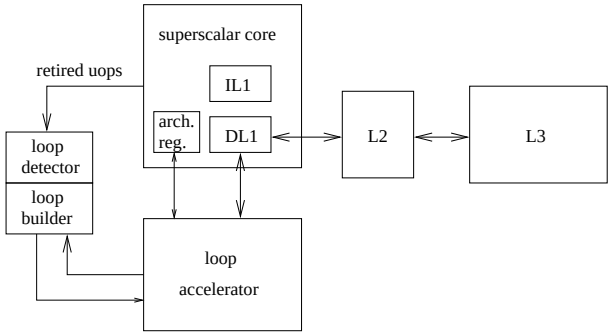

Figure 3: The superscalar core and the sequential loop accelerator

in succession, about one third of all the instructions executed belong to loops (as we have defined them) with a maximum body size of 256 instructions. In fact, many loops have a body not exceeding a few tens of instructions. Yet, several benchmarks spend a signicant part of the execution in loops whose body size exceeds 128 instructions, in particular 433.milc, 444.namd, 459.GemsFDTD, 465.tonto and 483.xalancbmk. We define a *loopy* benchmark as a benchmark with more than 20% of dynamic loop instructions with MBS=256. About half of the SPEC CPU2006 benchmarks are loopy according to this definition. Loopy benchmarks are listed in Table 2, along with the average loop length according to our loop detector. In the remaining, we assume MBS=128 instructions.

5 The loop accelerator

This study considers the possibility of accelerating the execution of dynamic loops, without help from the programmer and preserving binary compatibility. The goal of this study is not to explore the design space of loop accelerators (**LAs**), which we believe is huge. Instead, we have tried to imagine a new microarchitecture avoiding conventional superscalar techniques that do not scale well. We tried to exploit loop characteristics as much as possible in order to make the LA scalable. We have also tried to make the LA as general as possible so that it can accelerate loops with a small or large body, as it was shown in the previous section that both cases are important.

The proposed microarchitecture is depicted in Figure 3. We consider here only the "sequential" part of a heterogeneous many-core. During loop building, the loop builder takes as input the μops retired from the superscalar core reorder buffer and prepares the loop body for execution on the LA. When loop building is done, the superscalar core instruction window is cleared and the execution is migrated to the LA. When a loop exit condition occurs, like a branch taking a different direction, the architectural registers are updated and the execution is migrated back onto the superscalar core, until another dynamic loop is encountered.

The rest of this section is organized as follows. Section 5.1 introduces the cyclic register dependency graph, which is useful for characterizing a loop and understanding its execution on a LA. The proposed LA is described in Section 5.2 and Section 5.3 focuses on memory dependencies. Section 5.4 explains how the loop body can be reduced by removing redundant μops. Section 5.5 explains how we map the loop body

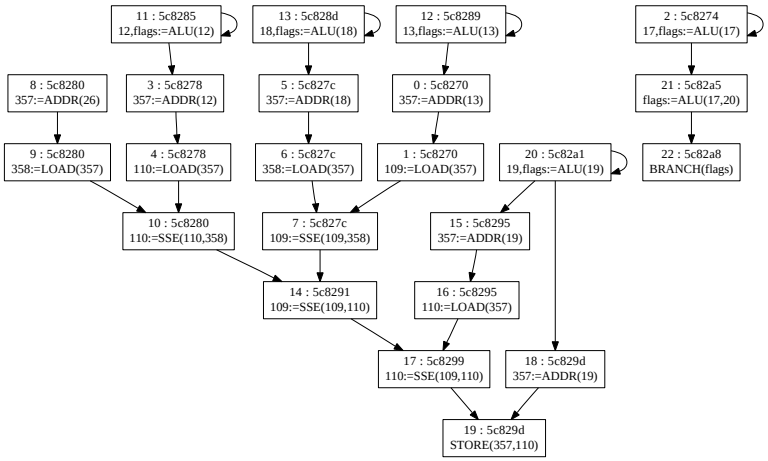

Figure 4: Example of loop CRDG found in benchmark 481.wrf. The body consists of 15 instructions split up into 23 static μops ranked from 0 to 22. Dependencies with μops not belonging to the loop are not part of the CRDG.

onto the LA for good performance. Other performance tuning we did is described in Section 5.6, and the simulated performance speedups are presented in Section 5.7.

5.1 Cyclic register dependency graph

To understand the performance of a LA, it is convenient to consider the *register dependency graph* (RDG) of a sequence of μops, where each node of the graph is a dynamic μop and directed edges represent register dependencies. The RDG of a loop has a periodic structure and it is possible to summarize it with a *cyclic register dependency graph* (CRDG) where each node of the graph is a static μop. Figure 4 shows a loop CRDG consisting of 23 static μops ranked from 0 to 22. The *rank* of a static μop corresponds to the sequential order. For example, the loop represented in Figure 4 consists of the dynamic instances of μops 0,1,...,22, 0,1,...,22,... and so on. Dependencies in a CRDG are of two sorts : *intra-iteration* and *inter-iteration*. When the μops producing and consuming a register value belong to the same iteration, it is an intra-iteration dependency. Otherwise it is an inter-iteration dependency. For an inter-iteration dependency, if the producer belongs to iteration k, the consumer belongs to iteration $k+1$. In Figure 4, inter-iterations dependencies are represented by edges whose head rank is less than or equal to the tail rank (e.g., both edges originating from μop 11). The longest chain in a RDG defines a minimum execution time. For a loop, the longest RDG chain can be found by considering cycles in the CRDG. Most loops have 1-μop cycles, like the loop shown in Figure 4 (μops 2,11,12,13,20 depend on themselves). However, a few loops have cycles consisting of several μops. The longest chain in a loop RDG corresponds to the greatest CRDG cycle, and the chain length is equal to the cycle size times the number of loop iterations. The actual execution time, though, may be longer than this minimum time if resources for executing the μops are limited. The impact of resource constraints on a loop performance can be taken into account by adding *constraint edges* to the CRDG. For instance, in order to simplify the hardware and permit a high clock frequency, we force the dynamic instances of the same static μop to execute in program order. This corresponds to adding to each μop in the CRDG a constraint edge to itself. Doing so does not increase the loop execution time. On the other hand, if we add a constraint edge between two distinct μops of the CRDG, we may create an artificial cycle consisting of several μops, thereby increasing the loop execution time.

5.2 A ring-shaped loop accelerator

Figure 5 depicts the global architecture of the proposed LA. The LA consists of 8 execution nodes (*node* for short) connected by a 4-lane pipelined bus. There are different node types, each node type being specialized for certain μops. The I-node executes μops having a 1-cycle latency. This includes ADDR μops, MOV μops (integer and floating-point) and all the μops that would normally execute on the superscalar core ALUs. The

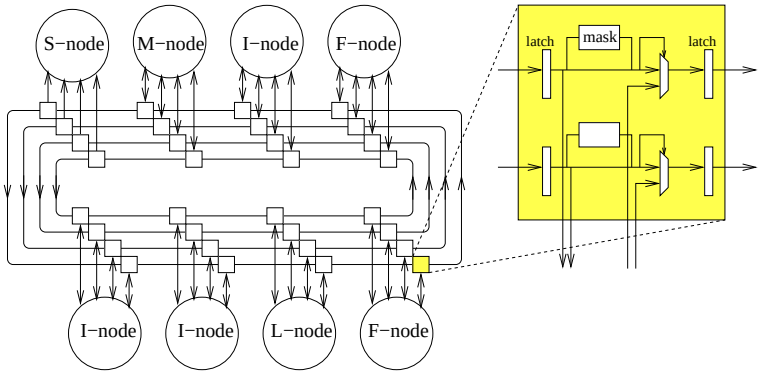

Figure 5: Ring-shaped loop accelerator : 8 execution nodes connected by a 4-lane pipelined bus.

F-node can execute the most frequent floating-point operations, in particular additions and multiplications. It executes integer multiplications too. The L-node executes load μops and the S-node executes store μops. The M-node is a *mutable* node whose function is defined at loop building. It is actually an I-node augmented with floating-point dividers. If the loop body contains some floating-point division, the M-node executes *only* divisions. Otherwise it behaves like a normal I-node. It is not necessary for the LA to be able to execute the whole instruction set. For instance, we noticed that integer divisions are rare in the loops found by our loop detector in the SPEC CPU2006. Therefore, loops featuring integer divisions are executed by the superscalar core.

The pipelined bus. The pipelined bus consists of 4 lanes divided in 8 *segments*. The bus transports *packets*. A packet consists of a 128-bit data [2], a μop rank, and an 8-bit *node vector* (one bit per node). The μop rank tells which μop in the body is the producer of the data. The node vector tells which nodes need the data. The node vector is modified by a mask as it passes through segments. When the packet reaches the last node needing the data, the node vector becomes null and the packet is considered empty. Now it is possible for that node (or a subsequent node) to overwrite the empty packet. In the worst case, a data does a complete turn on the lane until it returns to the node from which it was emitted. There are 2 latches per lane segment (i.e., 16 latches per lane). When the bus clock is low, the first latch is transparent and the second latch is opaque. When the bus clock is high, it is the other way around. Because of lane segment RC delays, we assume that the bus is clocked at half the node clock frequency [3]. To compensate this loss of throughput, each lane is doubled so that it transports two packets instead of one, as shown in Figure 5.

The execution node. Figure 6 shows the global structure of an I-node. Other node types share many characteristics with the I-node. Each node contains 4 *execution units* (**EU**). Each EU can execute 1 μop per cycle, i.e., the maximum node throughput is 4 μops per cycle. The loop builder maps static μops to EUs. All the dynamic instances of a static μop are executed by the same EU. **The μops executed by the same EU execute in program order with respect to each other**, but they can execute out-of-order with respect to μops on other EUs. Once a μop is executed, its output data is sent to the Output Queue (**OQ**), with the corresponding μop rank and node vector. A copy of the data is also pushed into the Loop Exit Queue (**LEQ**). The LEQ is used at loop exit time for updating the architectural registers with the correct values before resuming the execution on the superscalar core. Data in the OQ is sent onto the bus and removed from the OQ as soon as possible, i.e., when there is an empty packet at the corresponding lane segment. The execution of μops on a given EU is suspended if the associated OQ is on the verge of becoming full. At the node input, packets are read from the lanes. If the corresponding bit is set in the node vector, the data and μop rank are inserted into the Input Queue (**IQ**). We assume that **the IQ never gets full** (we will see later how we enforce this). Consequently, the data produced by a static μop enter the IQs where they are needed in sequential order. In each cycle, one data is dequeued from each IQ by the *distributor*. The distributor sends these data to the External Value Queues (**EVQ**). Each EU has a dedicated set of EVQs. A data from an IQ may be needed by several EUs in the same node. In this case

[2] We assume 128-bit data, as this is the data size required by Intel SSE operations. We did not try to optimize this. Future work may try to reduce the data size to 64-bit, exploiting the fact that most operations are actually scalar.

[3] It takes 16 LA clock cycles for a data to do a full turn on a lane.

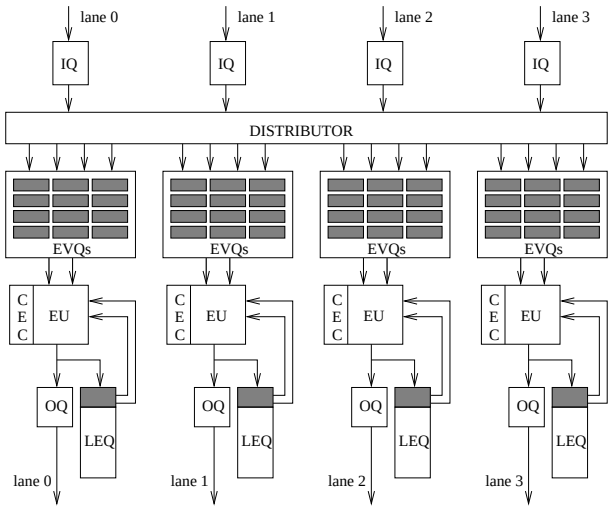

Figure 6: Execution node (here, an I-node). Each execution node contains 4 execution units.

the distributor duplicates the data (up to 4 copies, one per EU). An EVQ is associated with one particular static μop, i.e., all the data going through a given EVQ come from the same static μop. If a data cannot be distributed because one of the target EVQs is full, the data remains in the IQ until all the target EVQs can accept it. Non-constant data feeding the EU come either from an EVQ head or from the LEQ "top". The LEQ top consists of the N_{max} data output most recently by the EU, where N_{max} is the maximum number of static μops that can be mapped on the same EU (the LEQ is physically implemented as two distinct parts). The EVQ head data is dequeued only after the value is no longer needed by any μop on the EU. The execution of μops on an EU is controlled by a Cyclic Execution Controller (**CEC**). The CEC loops through the N_{eu} static μops mapped onto the EU, repeatedly and in program order. In particular, the CEC controls the various multiplexors for selecting the EU inputs and dequeues data from EVQs. The CEC also holds constant values, i.e., values that are used as input by some μops and that are guaranteed to be constant throughout the loop execution (e.g., register values not modified by the loop). The CEC generates a *sequence number* for each dynamic μop : the sequence number for the nth dynamic instance of a static μop of rank k is $B_{max} \times n + k$, where $B_{max} = 8 \times MBS$ is the maximum number of μops in a loop body [4] and n corresponds to the iteration number. Each EU executes μops in program order, which means that the sequence number increases monotonically. For μops with inter-iteration input dependencies, input data for the first iteration are obtained at loop building. It should be noted that there exists a data path from each EU to every other EU in the LA, thanks to the distributors. We have chosen not to have any direct data path between EUs in the same node to simplify the hardware [5]. Hence if a μop takes as input a data produced by another μop in the same node but on a different EU, this data must travel around the lane to reach the consumer μop. If a short communication latency between 2 μops is needed for performance, we must try to map these 2 μops on the same EU, so that the consumer μops can catch the data from the LEQ top. If we find during loop building that the loop cannot be executed by the LA, the loop keeps executing on the superscalar core. This happens for instance if the loop contains a μop that the LA cannot execute, like an integer division. This happens also if N_{max} is too small for that loop, or if there are too few EVQs. The F-node and M-node have a global structure similar to the I-node. The L-node and S-node are somewhat different.

The S-node. The S-node receives store addresses and store data from other nodes, it does not write on the lanes (no OQ, no LEQ). There is one Loop Store Queue (**LSQ**) associated with each of the 4 EUs. EUs do not really execute the store μops, they just gather the address and data for each dynamic store. The address and data are then sent into the LSQ, along with the store sequence number. The S-node contains a *store validation unit* (**SVU**). The SVU sends stores into the post-retirement store queue, which the LA shares with

[4] B_{max} is a power of 2, generating the sequence number is simple
[5] Future work may reconsider this choice.

the superscalar core and from which stores can write into the DL1 cache. The SVU has a table containing the ranks of all the static store μops in the loop body, in program order. The SVU logic loops through this table repeatedly, generating the sequence number corresponding to the next dynamic store, in program order. This SVU sequence numbers is searched among the 4 LSQ heads. The store is then *validated* : it is dequeued from its LSQ, sent to the post-retirement store queue, and the SVU sequence number is updated. In our simulations, we have assumed that the S-node can validate 2 stores per cycle.

The L-node. The L-node receives load addresses from I-nodes. It reads the DL1 cache and sends the load data through the bus to the other nodes. In the L-node, EUs are replaced with Load Issue Buffers (**LIB**) and OQs are replaced with Load Output Queues (**LOQ**). Each CEC steps through the static loads that have been mapped to it, in program order. The CEC allocates an entry in the LIB and in the LOQ for each dynamic load. The load address and LOQ entry identifier are written in the LIB entry. The LOQ entry has room for the load data. The CEC writes in the LOQ entry the μop sequence number and the 8-bit node vector for the data. In each cycle, one load is selected from each LIB and accesses the cache, i.e., the L-node can issue up to 4 loads per cycle. If the DL1 read succeeds, the load data is written into the LOQ entry, and the LIB entry is freed. Otherwise (bank conflict, DL1 miss, TLB miss), the load waits in the LIB and will be reissued later. Upon a miss, when the missing block is eventually inserted into the DL1, the associated MSHR wakes up the LIB entries that were waiting for that block, and the corresponding loads become ready for reissue. The LOQ behaves like a reorder buffer for loads. In a clock cycle, the load at the head of a LOQ is dequeued if the load data is there and if the packet on the lane segment can be overwritten.

The global synchronizer. The CEC has an N_{max}-entry μop buffer, a pointer on that buffer, and a Local Iteration Count (**LIC**). The pointer points to the μop that will next access the EVQs [6]. The pointer can be incremented in each clock cycle and is reset when its value becomes equal to N_{eu}. Every time the pointer is reset, the LIC is incremented. The 32 CECs (8 nodes, 4 lanes) work independently from each other. They may have different LIC values at a given time. However, we need a global *synchronizer* for keeping the program sequential semantic, a mechanism equivalent to the reorder buffer in a superscalar core.

In particular, the synchronizer must prevent a store from leaving the SVU before all the dynamic μops preceding the store in program order have been executed, as these μops may trigger a loop exit. A *natural* loop exit occurs when a static branch changes its behavior, i.e., a branch μop produces a result different from the one recorded at loop building [7]. Stores after the loop exit point must not write into the DL1. After the loop exit, the LEQs are used for updating the architectural registers. But the LEQ size is limited. Hence one of the synchronizer function is to prevent the execution on an EU from going too far ahead, making sure that the values needed for updating the architectural registers have not been pushed out of the LEQ. The synchronizer is connected to all the nodes, receiving and sending signals from and to the nodes. The SVU in the S-node maintains a *maximum sequence number* (**MSN**). If the loop contains any store, store validation freezes when it reaches the MSN, and the SVU sends a signal to the synchronizer. Moreover, each CEC has a maximum value **LICmax** for its LIC. Different CECs may have different LICmax values, but with some contraints : In the S-node, all the CECs have the same LICmax = **MinLICmax**, where MinLICmax is fixed at loop building. On the other nodes, LICmax $\geq$ MinLICmax. When the LIC in a CEC reaches MinLICmax, the CEC sends a signal to the synchronizer. The CEC continues until the LIC reaches LICmax, then the CEC freezes. When the synchronizer has received a signal from each CEC and from the SVU, all CECs have a LIC greater than or equal to MinLICmax. The synchronizer then sends an *unfreeze* signal to all nodes. Upon receiving the unfreeze signal, each CEC subtracts MinLICmax to its LIC, which unfreezes automatically the frozen CECs (e.g., those in the S-node). Upon receiving the unfreeze signal, the SVU adds MinLICmax $\times B_{max}$ to the MSN, which unfreezes store validation. The synchronizer maintains the program sequential semantic while preserving some decoupling between CECs. Still, it may impact performance. LICmax determines the instruction window size. The higher LICmax, the larger the instruction window and the more latency tolerance. However, the number of loop iterations in the window must not exceed the LEQ size, i.e., LICmax + MinLICmax must not exceed the LEQ size divided by N_{eu}. Global synchronization latency may also impact performance : the time during which a CEC is frozen is a waste of performance. Hence MinLICmax must not be too small. Another constraint is that MinLICmax cannot be greater than the LSQ size divided by N_{eu}, as the LSQ buffers the stores until the unfreeze signal increases the MSN, which permits validating the stores waiting in the LSQ. In summary, the LEQs and LSQs

[6] The execution is fully pipelined (except floating-point divisions in the M-node). Data dependencies may stall the execution.
[7] Exceptions are another loop exit condition.

must be large enough for good performance.
Loop exit. When the execution of a μop triggers a loop exit, the μop sequence number, which we call the loop exit sequence number (**LESN**), is sent to the synchronizer [8]. The LESN is then broadcast to all CECs. CECs whose sequence number already exceeds LESN freeze immediately and ignore subsequent unfreeze signals. Other CECs continue to work until exceeding the LESN. If a new loop exit is detected while a previous loop exit was already pending, and if the new LESN is less than the LESN recorded in the synchronizer, it becomes the new loop exit point. We can exit the loop acceleration mode when all CECs sequence numbers exceed the LESN, all EU pipes have beed drained, the SVU sequence number exceeds the LESN, and each LOQ in the L-node is either empty or has its head entry sequence number exceeding the LESN. Then, architectural registers can be updated with values found in the LEQs and the execution can resume on the superscalar core, starting from the first dynamic instruction following the loop exit point.
Full IQ and premature loop exit. We have assumed that the IQs never get full. It is actually difficult to make sure that an IQ can never get full. But is it possible to make it a rare event. If an IQ becomes full, we trigger a *premature* loop exit. The LESN is set to MSN-1. Then, the loop exit takes place as described previously. Nevertheless, for limiting the occurrence of premature loop exits without oversizing the IQs, we introduce a throttling mechanism. We associate a wired-OR with each lane. When the occupancy of any IQ on a lane exceeds a certain threshold (e.g., half the IQ capacity), the wired-OR is asserted and the OQs on that lane stop sending data until the wired-OR is deasserted.
Deadlocks. The LA we have described is not immune to deadlocks. Instead of trying to prevent deadlocks in all cases, it is simpler to make them as rare as possible. In particular, the EVQs must be large enough to maintain a low deadlock probability. We detect a deadlock when no unfreeze signal has been generated for 10000 cycles. When this happens, we trigger a premature loop exit. A deadlock leads to a much higher performance penalty than a full IQ.

5.3 Memory dependencies

The proposed loop accelerator can emulate a window of several thousands of instructions. But we must deal with memory dependencies. We must guarantee a correct execution without sacrificing too much performance. We describe in this section some mechanisms to deal with memory dependencies inside loops. We exploit loop properties to simplify memory dependency enforcement. First we expect most loops to exhibit a very good temporal and spatial locality . Second, we expect constant-stride accesses to be frequent in loops. Third, we expect dependencies between a store and a load in the same loop to repeat on each iteration.
The memory zone checker. The *memory zone checker* (**MZC**) is a table located in the L-node but accessed both by loads and stores. The purpose of the MZC is to detect memory order violations within loops. The MZC takes advantage of spatial locality that exists in loops. Our MZC is conceptually similar to the Memory Disambiguation Table described by Stone et al. [15] except that we record only loads in the MZC. We define a zone as a memory region whose size is a power of two and which is aligned in memory. The main originality of our MZC is that the zone size is fixed at loop building (cf. Section 5.6). The zone address is obtained from the load/store address by a right shifting of the address bits. Each MZC entry is tagged with a distinct zone address and contains a valid bit, a load sequence number, a load address, a load data size, a conflict bit, a load PC and a *known_pc* bit. When a load executes, it accesses the MZC with its zone address Upon a MZC miss, we search a free entry and we initialize it. A free entry is an entry whose valid bit was reset because its sequence number is less than the SVU sequence number. If no free entry is found, a premature loop exit is triggered. Upon a MZC hit, the sequence number in the MZC entry is compared with the load sequence number. If the load sequence number is greater, it overwrites the entry sequence number. If the data address and data size in the entry do not match those of the load, we set the conflict bit. If the load PC does not match the PC in the entry, we reset the *known_pc* bit. The MZC is also accessed by stores when they are validated by the SVU. A potential memory order violation is detected if the store zone hits in the MZC, if the store sequence number is less than the entry sequence number, and if the conflict bit is set or if the store data overlaps with the address and data size recorded in the entry. The optimal zone size, i.e., the one that minimizes the number of premature loop exits, is not the same for all loops. If there is a good spatial locality in the loop, taking a large zone prevents filling the MZC. On the

[8] The loop exit point must correspond to an architectural state, i.e, the loop exit μop must be the last μop of an instruction.

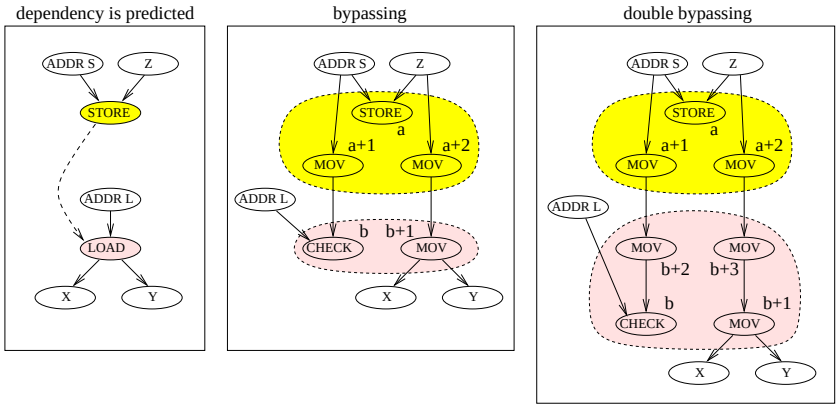

Figure 7: Store-to-load bypassing (a and b are the ranks of the store and CHECK μops after transformation)

other hand, if loads and stores are independent but access the same memory zones, decreasing the zone size may prevent unnecessary premature loop exits. In our simulations, we have assumed that the MZC could be updated by 4 loads and read by 2 stores in the same cycle.

The stride-based checker. The MZC is sufficient to guarantee a correct execution, it is not sufficient for good performance. A reasonably-sized MZC may trigger a lot of premature loop exits on inexistent memory dependencies. We introduce a *stride-based checker* (**SBC**) to assist the MZC. The SBC takes advantage of constant-stride memory accesses. The SBC is a small fully-associative table where each entry summarizes the memory accesses generated by a particular static load. Each SBC entry is tagged with the load PC and contains a valid bit, a sequence number, a *first* address, a *last* address, a stride, a data size, and an iteration count. When executing a load, we search that load PC in the SBC. If an entry exists, we update it as follows. If the iteration count is null, we initialize the entry with the load address, data size and sequence number. Else, if the iteration count is non-null, we compute $S = \text{load_address} - \text{last_address}$. If the iteration count equals 1, we set the stride to S. Otherwise, if the iteration count is greater than 1 and if the stride differs from S, we invalidate the entry. Eventually, we set the last address to the load address, and we increment the iteration count. When validating a store, if the MZC detects a potential memory order violation and if the *known_pc* bit is set, we access the SBC with the load PC recorded in the MZC entry. If the entry does not exist, we create it (possibly evicting a load), reset its content, and a premature loop exit is triggered with the store as the loop exit point. If the entry exists, we use its content to confirm the possible memory order violation. We use the first address, the last address, the stride sign and the data size to find which memory region the load has accessed so far. If the store data does not overlap with that region, we are sure that there was no memory order violation for that store. Otherwise, we assume that there was one, and a premature loop exit is triggered. When validating a store, we "remove" the oldest dynamic load of *every* valid SBC entry whose iteration count is non-null and whose sequence number is less than the store sequence number : we decrement the iteration count and, if the iteration count is non-null, we add B_{max} to the sequence number and we add the recorded stride to the first address.

Store-to-load bypassing. Some loops actually contain true memory dependencies. A solution for allowing the LA to execute these loops is to do store-to-load bypassing at loop building. The loop builder contains a Store Sets memory dependency predictor [2], identical to the one implemented in the superscalar core [9]. At loop building, if a static load is predicted to depend on a static store with the same data size, we transform the body as shown in Figure 7. The store μop stays in place, but 2 extra MOV μops are added in the loop body just behind the store [10]. The load μop is removed from the body and replaced with a CHECK μop and a MOV μop. The MOV μop transmits the store data to the μops consuming the load value. The CHECK μop compares the load and store addresses. All CHECK μops and MOV μops execute on I-nodes. If a CHECK fails, a loop exit is triggered, taking as loop exit point the dynamic μop preceding (in sequential order) the instruction containing the removed load.

[9] Our store sets LFST is fully associative, tagged with the SSID, which permits taking a wide SSID while keeping the LFST small.

[10] Several loads may be predicted to depend on the same store. the MOV μops for the store are generated only once.

The Bypassed Store Table. Even if a CHECK succeeds, we must still verify that stores between the dependent store-load pair do not write that memory location. The SVU contains a small fully-associative Bypassed Store Table (**BST**). Each BST entry contains a store address, a store PC, a data size and a *lifetime*. When validating a bypassed store, we search a free BST entry for that store, i.e., an entry whose lifetime is null. We initialize this entry with the store address, PC, data size, and we set the lifetime to the maximum number of dynamic stores separating the bypassed store-load pair(s). This value, which may be null, is obtained during loop building. If the lifetime is non-null and no free entry was found in the BST, we trigger a loop exit with that bypassed store as the loop exit point. When validating a store (whether bypassed or not), we check whether the store conflicts with any BST entry whose lifetime is non-null. If a conflict is detected, a loop exit is triggered with the validated store as the loop exit point. For each validated store, we decrement all the non-null lifetime values in the BST. When a conflict is detected, we train the store sets predictor with the validated store PC and the bypassed store PC. Doing so merges these two stores' store sets [2], so that on future occurrences of that loop, the load is predicted to depend on the correct store.
Double bypassing. The bypassing method described previously is effective only if the distance between the dynamic store and the dynamic load is less than one loop body. We found that this represents the majority of cases. However one benchmark, 456.hmmer, suffered many failed CHECKs. To solve this case, we introduce *double bypassing*, which is the possibility for a dynamic store to communicate its data to a dynamic load at a distance between one and two bodies. Double bypassing is illustrated in Figure 7. It uses the BST like simple bypassing, except that the BST entry lifetime is set to a longer value. The loop builder contains a Failed Check Table (**FCT**), which his a small fully associative table. When a CHECK triggers a loop exit, we record in the FCT the bypassed load PC, so that if a predicted-dependent load hits in the FCT during loop building, we apply double bypassing.

5.4 Loop body reduction

Redundant execution exists in most programs [13]. On the example of Figure 4, μops 8 and 9 are redundant, they produce the same result on each iteration. This result is obtained at loop building, hence these μops can be removed from the loop body before executing the loop on the LA. This is an iterative process : in each iteration during loop building, we remove from the CRDG the μops that have no inputs left in the CRDG. The number of redundant μops identified depends on LBI [11]. We must be careful with loads and stores. A store μop, even if it produces the same result on each iteration, cannot be considered redundant in a shared-memory architecture, as removing it would break the memory consistency model. A redundant load can be removed, but we must check memory dependencies. Some hardware support is necessary for that. The SVU contains a Removed Loads Table (**RLT**). Each RLT entry contains a valid bit, a load address, a load data size and a load PC. During loop building, as long as there is room in the RLT, redundant loads are removed from the body and recorded in the RLT. When validating a store, we check whether the store conflicts with any RLT entry. If a conflict is detected, a loop exit is triggered with the store as the loop exit point. The PC recorded in the conflicting RLT entry is used to train the store sets predictor. For loopy benchmarks, we found that redundant μops represents typically between 10% and 20% of dynamic loop μops.
MOV bypassing. It is possible to bypass certain MOV μops, meaning that, at loop building we make the MOV's consumer μop depend directly on the MOV's producer μop. MOV bypassing is possible if the distance (in the sequence of dynamic instructions) between the producer and consumer μops is less than one loop body. If MOV bypassing can be applied for all the consumers of the MOV, the MOV itself can be removed from the body [12].

5.5 Mapping heuristic

The LA performance is very dependent on the mapping of μops onto EUs. The in-depth study of mapping heuristics is left for future studies. For this preliminary study, we tried to find a good-enough mapping through trial and error. The heuristic we found helped us understand some important properties for a good mapping heuristic on such LA. We just give a high-level description here, omitting some details.

[11] With LBI=5, we were able to remove all the redundant μops.
[12] Solutions exist for bypassing all MOVs, this is left as future work.

clock frequency : 3 GHz ; **LM table** : 4 entries ; **LT** : 64 entries ; **MinLM** : 900 instructions ; **LBI** : 5 iterations; **MBS** : 128 instructions ; $\mathbf{N}_{max}$: 12 μops; **EVQs per EU** : 12 ; **EVQ** : 32 data ; **IQ** : 32 data ; **OQ** : 16 data ; **LIB** : 64 loads ; **LOQ** : 128 data ; **LEQ** : 128 data ; **LSQ** : 64 stores ; **MZC** : 64 entries ; **SBC** : 16 entries ; **BST** : 16 entries ; **FCT** : 16 entries ; **RLT** : 64 entries ; **SVU throughput** : 2 stores / cycle ; **unfreeze signal latency** : 4 cycles ; **wired-OR latency** : 4 cycles ; **loop mode transition penalty** : 100 cycles ; **extra transition penalty on a LT miss** : 10000 cycles ; **store sets SSIT** : 4096 entries ; **store sets LFST** : 16 entries ; **SSIT clearing period** : 10 millions cycles ;

Table 3: Fixed parameters for the loop detector, loop builder, and loop accelerator.

EU balancing. The μops mapped onto the same EU form a cycle in the constraint-augmented CRDG. Therefore, the average number of clock cycles per loop iteration cannot be less than the N_{eu} of the most loaded EU. The mapping heuristic must try to reach a balanced distribution of μops on EUs. This is particularly important for loops with a small body : the mapping heuristic should avoid putting two μops on the same EU whenever possible, as this potentially doubles the loop execution time. Our mapping heuristic computes, from the body characteristics, a μop quota per EU type, assuming EU balancing. Once a μop is mapped onto a EU, we consider another μop, and so on until all μops have been mapped. We try to avoid mapping a μop onto an EU that has already reached its quota.
Lane segment sharing. Data that must go through the same lane segment share its bandwidth, which we have limited to one data per LA clock cycle. The μops producing these data cannot execute at an average rate greater than the rate at which the data go through the shared lane segments. Our mapping heuristic tries to minimize lane usage but does not try explicitly to minimize lane segment sharing [13]. We try to map a μop on the same EU as one of its input μops, provided this EU has not reached its quota. Many μops have a single consumer, and this often permits avoiding using the bus for transmitting the data. If we must use the bus, we try to put the consumer μop on the node following the node where the producer μop has been mapped, whenever possible.
Natural CRDG cycles. The CRDG contains some *natural* cycles due only to data dependencies. Most natural cycles are 1-μop cycles consisting of an integer addition depending on itself. Yet, some loops contain multi-μop cycles which may increase the loop execution time considerably, as it takes several cycles for a data to travel from one EU to another. Whenever possible, our mapping heuristic tries to map onto the same EU the μops forming a natural cycle.
Artificial CRDG cycles. Artificial CRDG cycles are cycles in the constraint-augmented CRDG that are not natural cycles. Artificial CRDG cycles may increase the loop execution time dramatically, either because the cycle contains floating-point operations or, worse, because some μops in the cycle are mapped onto different EUs. Mapping μops onto the same EUs as their producers permits decreasing the probability of creating costly CRDG cycles, but this is not always sufficient. Whenever possible, our mapping heuristic tries to avoid mapping a μop on an EU if this would create an artificial CRDG cycle.

5.6 Performance tuning

Our simulator implements the loop detector and loop accelerator as described in Sections 4 and 5, i.e., with a great level of detail. Parameters we have used for the simulation are given in 3. Notice the loop mode transition penalty, that we have fixed at 100 clock cycles. We assume that this penalty takes into account the time elapsed after loop building and before the LA starts executing the loop, and the time necessary to update the architectural registers after the loop exit. In case of a LT miss, we add an extra penalty of 10000 cycles.
Maximum LICmax. For good performance, the values of LICmax and MinLICmax are set dynamically at loop building. As explained before, we try to set LICmax and MinLICmax as high as possible but under the limit permitted by the LSQ and the LEQ (which also depends on N_{eu}). However, when LICmax exceeds the EVQ depth, the deadlock probability becomes non-null. Actually, there is a LICmax value beyond which the deadlock probability increases dramatically. This value is not the same for all loops. To solve this issue,

[13] This is a possible area of improvements.

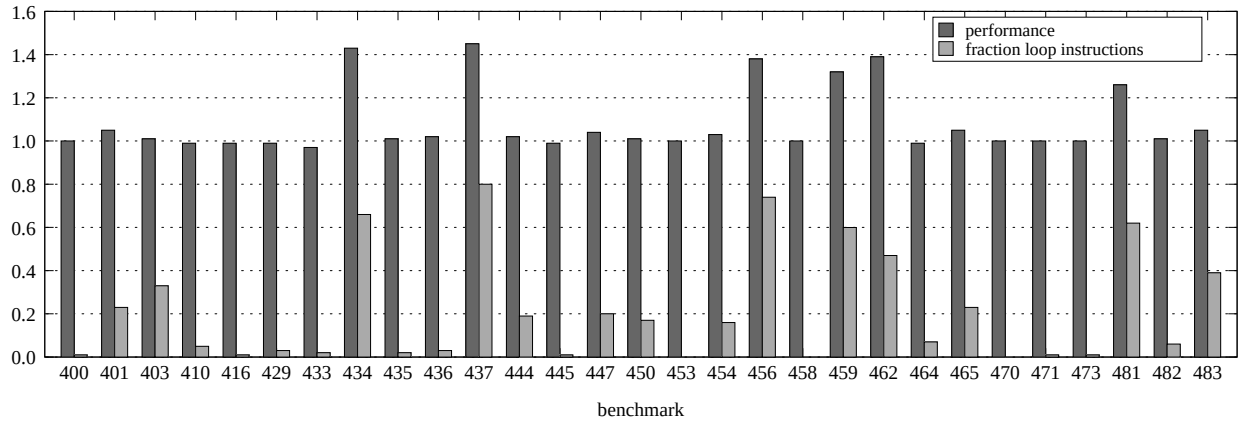

Figure 8: Global performance speedups obtained with the loop accelerator, relative to the baseline with no accelerator (higher is better). The second bar represent the fraction of instructions executed by the loop accelerator.

we record in each LT entry a value **ML** which defines a maximum for LICmax. When a deadlock occurs, we halve the ML value recorded in the LT entry for that loop. Moreover, a high LICmax is useful only for loops with a short body, for which a large instruction window represents many iterations. If the body exceeds 20 instructions, or if the time elapsed since the last deadlock is less than 1 million cycles, we set ML equal to the EVQ depth. Otherwise, we allow ML to be as high as 128. Overall, with this method, deadlocks have a negligible impact on performance.

Memory zone size. The optimal memory zone size is not the same for all loops. We record in each LT entry the $\log_2$ of the zone size. Each LT entry also contains a 3-bit saturating counter. The zone size for a loop is initially set to 1024 bytes. When a premature loop exit occurs because the MZC is full, the 3-bit counter is incremented. If the 3-bit counter value is equal to 7, we double the zone size. If the MZC or the SBC signals a memory order violation, we decrement the 3-bit counter. If the 3-bit counter value is null, we halve the zone size.

Disabling "bad" loops. Migrating the execution to the LA may actually decrease performance. This generally happens because of premature loop exits. For good performance, "bad" loops must be detected and their execution on the LA disabled. Each LT entry contains a $TimeLost$ value which estimates the time that has been lost so far by executing the loop on the LA rather than on the superscalar core. On a loop exit, the loop detector is trained with the number n_i of instructions executed by the LA, the time n_t spent on the LA and the number n_m of L3 cache misses generated by the loop. The $TimeLost$ value is updated as follows : $TimeLost \leftarrow TimeLost + n_t - 0.7 \times n_i - 20 \times n_m$ where values 0.7 and 20 were found empirically. $TimeLost$ is initially set to 0. For most loops it becomes negative, indicating that the LA is likely to perform better than the superscalar core. For some loop, $TimeLost$ increases. When $TimeLost$ exceeds 100000 cycles, we consider that this is a "bad" loop. Sometimes, we reset all the $TimeLost$ values in the LT. We do this when the time elapsed since the last reset exceeds 100 times the sum of positive $TimeLost$ values in the LT.

5.7 Simulations results

Figure 8 shows the performance obtained with the loop accelerator. Speedups are relative to the simulated baseline (cf. Figure 1). The second bar in Figure 1 shows the fraction of instructions executed by the LA. Some speedups are obtained on most loopy benchmarks, except 433.milc, whose loop behavior comes mostly from loop bodies greater than 128 instructions. Performance gains exceeding 25% are obtained on 6 benchmarks : 434.zeusmp, 437.leslie3d, 456.hmmer, 459.GemsFDTD, 462.libquantum and 481.wrf. These 6 benchmarks are the ones that execute at least 60% of the instructions on the LA. We measured the local acceleration provided by the LA on the loop fraction. For the 6 benchmarks mentioned above, the local acceleration is respectively 2.3, 2, 1.7, 2.8, 2.1 and 1.9 (ignoring the transition penalty). This level of local acceleration would be difficult to obtain with conventional superscalar techniques.

	all	**no SBC**	**no byp.**	**no reduc.**	**naive mapping**	**ML = 32**	**zone = 1 KB**	**bad loops**
434.zeusmp	1.43	1.32	1.33	1.18	1.00	1.42	1.25	1.43
437.leslie3d	1.45	1.26	1.43	1.10	0.95	1.44	1.40	1.38
456.hmmer	1.38	0.99	0.99	1.22	0.99	1.38	0.99	1.38
459.GemsFDTD	1.32	1.00	1.13	1.13	1.00	1.33	1.28	1.32
462.libquantum	1.39	1.24	1.39	1.37	1.19	1.19	1.39	1.39
481.wrf	1.26	1.12	1.21	1.19	0.95	1.26	1.26	1.26
29 bench. mean	1.084	1.036	1.055	1.045	0.996	1.068	1.061	1.072

Table 4: Performance relative to the baseline. The second column is for all features enabled. Following columns give performance when disabling a single feature, respectively, disabling the SBC, store-to-load bypassing, loop body reduction, using a naive mapping heuristic, using a small ML, using a fixed zone size, and allowing bad loops.

Table 4 shows performance for the 6 benchmarks mentioned above and the average performance on all 29 benchmarks. The second column is for all features enabled. Following columns give performance when disabling a single feature. The mapping heuristic is very important. To quantify its impact, we have simulated a "naive" mapping heuristic which scans all EUs in a fixed order : we map on the current EU one μop that can be mapped on it, then we move to the next EU. We do this repeatedly until all μops have been mapped. With this naive heuristic, the average performance is even slightly lower than the baseline. Another important feature is the SBC. Without it, false memory dependencies trigger many premature loop exits. Loop body reduction is also very important. Without it, the loop accelerator is clearly not working at its full potential. Store-to-load bypassing brings significant performance gains on 456.hmmer and 459.GemsFDTD. The last 3 columns show that the performance tuning described in Section 5.6 bring non-negligible performance gains. The fixed memory zone size prevents 456.hmmer to benefit from the accelerator. Using a large ML is important for 462.libquantum, which has small loop bodies. Disabling bad loops is a useful feature, especially for benchmarks which do not benefit from the accelerator.

6 Conclusion and future work

We have proposed a hardware mechanism for detecting dynamic loops. We found that about one third of all the instructions executed by the SPEC CPU2006 suite come from dynamic loops whose body size can be quite diverse, from a few instructions to several hundreds.

We have proposed a loop accelerator microarchitecture that can accelerate most dynamic loops without help from the compiler or the programmer. The accelerator configuration we have focused on has 8 nodes and 4 lanes, and can execute up to 32 μops simultaneously from a window of several thousands of μops. Its local hardware complexity is no greater than that of a conventional superscalar core. Our efforts for obtaining good performance speedups have shown the importance of mapping μops onto execution units very carefully. We have proposed new solutions for dealing with memory dependencies in a window of several thousands of instructions, exploiting loop properties. We have shown that a significant fraction of dynamic loop instructions are redundant and need not be executed by the loop accelerator.

This is a preliminary study, intended to provide a basis for future work on loop acceleration. The design space of loop accelerators is huge we believe. The mapping heuristic is very important for performance. Some of the characteristics we have outlined for a good mapping heuristics are somewhat general we believe, like the importance of preventing artificial CRDG cycles. Other aspects of our mapping heuristic may be dependent on some of the choices we made, like not providing any direct data path between EUs on the same node, or choosing a pipelined ring for communicating between nodes. Future studies may reconsider these choices and try to find a better tradeoff between the IPC, the hardware complexity of the bus and nodes, and the mapping heuristic. Nevertheless, we believe that the loop accelerator microarchitecture we have proposed is scalable beyond the particular configuration we considered. For instance, increasing the number of I-nodes and F-nodes will not increase the local hardware complexity. Increasing the number of lanes

will increase the complexity of the distributor and some parts of the L-node (for instance the MZC, whose number of ports is proportional to the number of lanes). Yet, we believe that if one can double the width of the superscalar core, doubling the number of lanes should not be more difficult. The node microarchitecture depicted in Figure 6 is, we believe, easier to pipeline than a conventional 4-way superscalar execution core. Future work may consider the possibility to overclock the loop accelerator. The global speedups we have demonstrated are limited by Amdahl's law, i.e., by the fraction of the execution spent in dynamic loops. Nervertheless, the local acceleration we have obtained on dynamic loops is considerable for a hardware-only solution. A compiler aware of the presence of a loop accelerator may try to exploit it.

References

[1] T. W. Barr, A. L. Cox, and S. Rixner. Translation caching: Skip, don't walk (the page table). In *Proc. of the 37th Int. Symp. on Computer architecture*, 2010.

[2] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *Proc. of the 25th Int. Symp. on Computer Architecture*, 1998.

[3] N. Clark, A. Hormati, and S. Mahlke. VEAL : virtualized execution accelerator for loops. In *Proc. of the 35th Int. Symp. on Computer Architecture*, 2008.

[4] M. Dixon, P. Hammarlund, S. Jourdan, and R. Singhal. The next-generation Intel Core microarchitecture. *Intel Technology Journal*, 14(3), 2010.

[5] A. García, O. J. Santana, E. Fernández, P. Medina, and M. Valero. LPA : a first approach to the loop processor architecture. In *Proc. of the 3rd Int. Conf. on High Performance Embedded Architectures and Compilers*, 2008.

[6] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41(7):33–38, July 2008.

[7] M. Kobayashi. Dynamic characteristics of loops. *IEEE Transactions on Computers*, c-33(2):125–132, 1984.

[8] G. H. Loh, S. Subramaniam, and Y. Xie. Zesto : a cycle-level simulator for highly detailed microarchitecture exploration. In *Proc. of the Int. Symp. on Performance Analysis of Systems and Software*, 2009.

[9] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa Reddi, and K. Hazelwood. Pin : building customized program analysis tools with dynamic instrumentation. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005. http://www.pintool.org.

[10] M. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely Jr, and J. Emer. Adaptive insertion policies for high performance caching. In *Proc. of the 34th Int. Symp. on Computer Architecture*, 2007.

[11] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith. Trace processors. In *Proc. of the 30th Int. Symp. on Microarchitecture*, 1997.

[12] A. Seznec and P. Michaud. A case for (partially) tagged geometric history length branch prediction. *Journal of Instruction Level Parallelism*, April 2006. http://www.jilp.org/vol8.

[13] A. Sodani and G. S. Sohi. Dynamic instruction reuse. In *Proc. of the 24th Int. Symp. on Computer Architecture*, 1997.

[14] G. Stitt, R. Lysecky, and F. Vahid. Dynamic hardware/software partitioning : a first approach. In *Proc. of the Design Automation Conference*, 2003.

[15] S. S. Stone, K. M. Woley, and M. I. Frank. Address-indexed memory disambiguation and store-to-load forwarding. In *Proc. of the 38th Int. Symp. on Microarchitecture*, 2005.

[16] J. M. Tendler, J. S. Dodson, J. S. Field, H. Le, and B. Sinharoy. POWER4 system architecture. *IBM Journal of Research and Development*, 46(1), January 2002.

[17] J. Tubella and A. González. Control speculation in multithreaded processors through dynamic loop detection. In *Proc. of the 4th Int. Symp. on High-Performance Computer Architecture*, 1998.

[18] S. Vajapeyam, P. J. Joseph, and T. Mitra. Dynamic vectorization : a mechanism for exploiting far-flung ILP in ordinary programs. In *Proc. of the 26th Int. Symp. on Computer Architecture*, 1999.

[19] S. Vajapeyam and T. Mitra. Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences. In *Proc. of the 24th Int. Symp. on Computer Architecture*, 1997.

$\mathbf{B}_{max}$: maximum number of μops for the loop body ; **BST** : bypassed store table ; **CEC** : cyclic execution controller ; **CRDG** : cyclic register dependency graph ; **EU** : execution unit ; **EVQ** : external value queue ; **FCT** : failed check table ; **IQ** : input queue ; **LA** : loop accelerator ; **LBI** : loop build iterations ; **LEQ** : loop exit queue ; **LESN** : loop exit sequence number ; **LIB** : load issue buffer ; **LIC** : local iteration count ; **LM** : loop monitor ; **LOQ** : load output queue ; **LSQ** : loop store queue ; **LT** : loop table ; **MBS** : maximum body size in instructions ; **MinLM** : LM threshold in instructions ; **ML** : maximum value of LICmax ; **MSN** : maximum sequence number ; **MZC** : memory zone checker ; $\mathbf{N}_{eu}$: number of static μops mapped onto a given EU ; $\mathbf{N}_{max}$: maximum number of static μops per EU ; **OQ** : output queue ; **RDG** : register dependency graph ; **RLT** : removed loads table ; **SBC** : stride based checker ; **SVU** : store validation unit ;

Table 5: Acronyms and definitions

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399